Iceboot Users Guide

Arthur Jones
Lawrence Berkeley National Labs

August 20, 2003

The work described here was created with the help of many people – I'd like to point out just a few ...

Thanks to Azriel Goldschmidt, who patiently puts up with the buggy software and has offered many helpful suggestions and comments.

Thanks to Simon Patton, who put together the build environment which helps keep the project on track and usable by others.

Thanks to Thorsten Stezelberger, without whom I would have been completely incapable of testing any of the software described below.

Thanks to Chuck McParland, who pretty much completely defined what the software should do – definitely the hardest part of getting the software to work.

Special thanks to Gerald Przybylski for sharing his workspace the last few months and putting up with – even answering – so many of my naïve hardware questions.

Contents

1 Introduction

This document describes the DOM mainboard bootloader software environment for Iceboot users and programmers. While this document does not go into the details of the software design, it does cover the software at a high level, and should be a good place to get started working with Iceboot.

Iceboot – like most bootloaders – is a program for bringing up hardware from power off into a known and consistent state. The program must be able to execute programs (usually operating systems) and, at least at a simple level, must be able to read from filesystems.

As we worked with the DOM hardware we found that we would like a few additional features from our bootloader:

- Complete diagnostics, things don't always work perfectly the first time. It's nice to be able to see at a glance exactly where things are failing.
- Configurability, we want to be able to quickly choose configurable hardware parameters (CPU clock speed, SDRAM clock speed, etc. . .).
- Expressiblity, we want to be able to do a lot without having to resort to the time-consuming cycle of C program development.

We chose forth for our boot environment as it is small, interpreted and yet capable enough for the task at hand. Of course, we weren't the first to choose forth for out boot environment, see e.g. `www.openfirmware.org`

2 Booting the Hardware

After power on reset, the Altera Excalibur ARM processor begins the boot process. If the Boot from Flash pin is active, the Excalibur starts execution from the start of the EBI0 address space – hardwired to be at address 0x40000000. If the Boot from Flash pin is inactive, the CPU will take boot code and an fpga design file from a local programmable ROM device. This document describes the Boot from Flash mode, the boot from serial mode (ConfigBoot) will be described elsewhere. The CPU must initialize many pieces of hardware, so that the software environment can be setup. All of the hardware described below is configurable, a description of the configuration file used is found in section 4.3 on page 22. The following sections will step through the boot process.

2.1 Hardware Initialization

First in line to be programmed are the Excalibur PLLs. There are two of them: PLL1 and PLL2, PLL1 drives the AHB1 after a divide by two (AHB1 is one half the frequency of PLL1) and PLL2 drives the AHB2 clock after a divide by two. The AHB1 clock is the main CPU clock and the AHB2 clock is used for the SDRAM clock.

After the PLLs are setup, the iocontrol registers are configured, this lets us configure slew rates and other parameters for output pins on the Excalibur.

Next in line is initializing the UART. Here we setup the Excalibur chip to do serial communications. The chip is reset and the baud rate and other programmable options are setup.

After the UART is setup, the EBI bus parameters are setup. Generic options which affect all the EBI chip select devices are setup here, like clock rate, and also chip select specific options are setup here as well.

The SDRAM controller is configured next. The parameters for the SDRAM chip on the board are programmed into the controller and the controller is initialized.

Many of the devices that were configured can be accessed through memory locations on the Excalibur chip. These physical memory locations are setup next.

At this point in the bootup process, we have all the one-time hardware initializations done. We now want to load our boot code from flash to the SDRAM that has just been configured. After copying the boot code to the start of SDRAM, it is verified against the flash memory. If the verify fails, the booter will print out the expected value (flash value), the actual value read from SDRAM and the address of the error. These messages will continue in this format until all of the loaded image in SDRAM is checked.

The final step in the low-level boot code is to jump to the code that contains the boot prompt, Iceboot.

3 Iceboot Command Line Interface

3.1 Overview

Once the hardware is initialized, we want to run the command line boot environment, this will allow us to read and write memory locations, store and load images from the flash chips, execute programs stored in memory and other low-level functions. The booter provides a prompt which allows the user to type in commands one line at a time. If the user has access to a vt100 terminal, the Up-arrow and Down-Arrow keys will cycle through the last few commands run, the left and right arrows allows the user to position the cursor and the backspace key allows the user to destructively delete the previous character on a line. On most any terminal the Control-F and Control-B key sequences allow the user to go forward and backward one word. Iceboot implements an extremely limited subset of the Forth programming language.

3.1.1 Flash Filesystem

The DOM board has a very simple flash filesystem for persistant data storage. There is only one directory, and each file takes up contiguous blocks on the flash. The *ls* command lists the contents of the flash filesystem, the *find* command searches the directory, the *create* command creates a new file on the filesystem,

the *lock* command locks a file on the filesystem and the *unlock* command unlocks a file on the filesystem. Files are locked by default and *create* is the only command which automatically unlocks files before creating them. The first two files on the flash filesystem are reserved, they can be overwritten using the *create* command but they may not be removed.

3.1.2 Forth

Forth is a stack oriented language, most commands work on data on the stack and return their results to the stack. This makes entering commands a bit funny at times, but it makes the parser very easy to write, and therefore very small, fast and reliable. Here are a few references to forth material on the web:

- General forth info at `http://www.forth.org/`
- gforth at `http://www.jwdt.com/~paysan/html/gforth.html`
- Book at `http://www.forth.com/Content/Handbook/Handbook.html`

All commands are read one line at a time and so many commands *must* be on one line to be syntactically correct. The commands are described using the following syntax:

Description

arg1 arg2 sample *ret1 ret2*

Sample Command

The command word is sample, arg1 and arg2 must be put on the stack before sample is called and ret1 and ret2 are returned on the stack. so, e.g.:

Examples

```
> 1 2 sample .s
<2> 3 4
```

arg1 is 1 arg2 is 2 the command is sample and it returns ret1 which is 3 and ret2 which is 4 (the .s command proings the contents of the stack).

The rest of this section contains descriptions of the real forth words that are implemented by iceboot:

3.2 !, w!, c!

Description

value address !, w!, c!

Put value into address, 32bit, 16bit or 8bit move.

Examples

> `$1 $50000009 c!`

Enable to high voltage base by writing 1 hex (8 bits) into address $50000009.

> `&10 base !`

Set number base back to decimal.

3.3 *

Description

*a b * result*

a * b

Examples

```
> 2 4 * .
8
```

Multiple 2 and 4 to get 8.

3.4 +

Description

a b + result

a + b

Examples

```
> 2 4 + .
6
```

Add 2 and 4 to get 6.

3.5 -

Description

a b - result

a - b

Examples

```
> 4 2 - .
2
```

Subtract 2 from 4 to get 2.

3.6 .

Description

.

Print top of stack

Examples

```
> 1 .
1
```

Print a 1, the contents of the top of the stack

3.7 .s

Description

.s

Print entire stack

Examples

```
> 1 2 3 .s
<3> 1 2 3
```

Print 1, 2, 3 the contents of the entire stack

3.8 /

Description

a b / result

a / b

Examples

```
> 6 2 / .
3
```

Divide 6 by 2 to get 3

3.9 : name words ;

Description

: name words ;

Function definition. The function is stored with the name name and words will be executed in place of the name

Examples

```
> :  hi s" hi there" type crlf type ;
> hi
hi there
```

Print ”hi there” and carraige return line feed to stdout

3.10 <

Description

a b < *result*

a < b

3.11 =

Description

a b = *result*

a = b

3.12 >

Description

a b > *result*

a > b

3.13 ?DO words LOOP

Description

end start ?DO words LOOP

Execute code from end (exclusive) to start (inclusive). Words are executed end - start times and the variable *i* is set to the current loop count.

Examples

```
> 5 0 ?DO i LOOP .s
<5> 0 1 2 3 4
```

Put the numbers 0-4 on the stack.

3.14 @, w@, c@

Description

address @, w@, c@ *value*

Get a 32bit word, 16bit word or 8 bit word from address

3.15 \\

Description

\\

Comment character, all characters to end of line are ignored by interpreter

3.16 allocate

Description

nbytes allocate *address*

Allocate nbytes on the heap address is put on the stack.

Examples

```
> 4 allocate constant p
> 1 p !
```

Allocate 4 bytes on the heap, set the address to the constant p and put a 1 in that address

3.17 analogMuxInput

Description

channel analogMuxInput

Select analog mux input channel.

Channel	Signal
0	Toyocom
1	40MHz square wave
2	PMT LED current
3	Flasher board LED current
4	Upper Local Coincidence
5	Lower Local Coincidence
6	Communications ADC
7	Front End Pulser

The input mux channels are described in more detail at
`http://rust.lbl.gov/~gtp/DOM/API/DOM_CPLD_API_v1.0.html`

3.18 and

Description

a b and *result*

a and b

3.19 base

Description

base *address*

Address of number base (10=decimal, 16=hex, ...)

3.20 constant

Description

a constant *name*

Set name to constant value a

3.21 cp

Description

address length cp *cpAddress cpLength*

Copy a memory block to sdram – usually used to copy from flash to sdram.

Examples

```
> stest.sbifind if cp stest2.sbicreate endif
```

Copy the file in flash to memory, then create a new file on the flash filesystem with the same contents but a different name.

3.22 create

Description

dataAddress dataLength nameAddress nameLength create

Create a file on the flash filesystem with name at address nameAddress of length nameLength using data at dataAddress with length dataLength.

3.23 crlf

Description

crlf *stringAddress stringLength*

Push the carraige return line feed character address and length to the stack

3.24 disableCurrents

Description

disableCurrents *result*

address of flag to set to disable printing of currents on the top of the screen

3.25 disableHV

Description

disableHV

Disable PMT high-voltage base

3.26 domid

Description

domid *address length*

get address and length of boardID string

Examples

```
> crlf domid type type
7383382234da2
```

Print the dom id string

3.27 drop

Description

drop

Drop the top stack element

3.28 dup

Description

a dup *a a*

Duplicate the top stack element

3.29 enableHV

Description

enableHV

Enable PMT high-voltage base

3.30 exec

Description

execAddress execLength exec

Execute the program at execAddress with length execLength

Examples

```
> s" stf" find if exec endif
```

If the file named stf is found on the flash filesystem, exec it (run it).

3.31 false

Description

false *false*

return false (0)

3.32 find

Description

nameAddress nameLength find *fileAddress fileLength status*

find the file with name at nameAddress of length nameLength and return it's address and length to the stack if status=1, if status=0 then do nothing.

Examples

```
> s" stf" find if exec endif
```

Execute the file stf on the flash filesystem

Examples

```
> s" startup.fs" find if interpret endif
```

Interpret the contents of the file startup.fs on the flash filesystem

3.33 fpga

Description

sbiAddress sbiLength fpga *status*

Program the fpga with the sbi file at sbiAddress with length sbiLength, return status = 0 if all is ok, otherwise return error code.

Examples

```
> s" stf.sbi" find if fpga endif
```

Program the fpga with the sbi file named stf.sbi in the flash filesystem

3.34 free

Description

address free

Free allocated data at address.

3.35 gunzip

Description

dataAddress dataLength gunzip *destAddress destLength*

Unzip a gzip file at dataAddress of length dataLength and put the resulting data in destAddress with length destLength. if destLength=0 and destAddress=0 there was an error.

3.36 i

Description

i *value of counter*

returns value of index variable in a ?DO loop.

Examples

```
> 4 0 ?DO i LOOP .s
<4> 0 1 2 3
```

Put the numbers 0-3 on the stack.

3.37 if cmds endif

Description

cond if cmds endif

execute commands if cond is non-zero

Examples

```
> 3 readADC 1000 > if s" high current on +5V" type crlf type
endif
```

Read the +5V current and print a warning if it is too high.

3.38 init

Description

init

initialize flash filesystem, all data on flash will be *erased*

3.39 install

Description

install

install a new flash filesystem over the wire, all data on flash will be *erased* – and replaced with the data in the intel hex format file which is sent over the wire.

3.40 interpret

Description

srcAddress srcLength interpret

Interpret data at address srcAddress with length srcLength

Examples

```
> s" hi there type crlf type" interpret
hi there
```

Interpret the string given.

3.41 lock

Description

nameAddress nameLength lock

lock flash file with name at nameAddress with length nameLength

3.42 ls

Description

ls

print listing of flash filesystem directory

3.43 lshift

Description

value shift lshift *result*

result = value shifted left shift

3.44 not

Description

a not *result*

result = !a

3.45 od

Description

count address od

print count hex words at address

3.46 prtCookedCurrents

Description

prtCookedCurrents

show cooked currents (efficiency corrected)

3.47 prtPLL

Description

n prtPLL

print pll information for pll n (1 or 2)

3.48 prtRawCurrents

Description

prtRawCurrents

print raw currents (not efficiency corrected)

3.49 prtTemp

Description

code prtTemp

decode temperature and print it

3.50 rawCurrents

Description

rawCurrents *address*

return address of a flag to determine whether or not to print raw or cooked currents to the top of the screen.

3.51 readADC

Description

channel readADC *value*

read ADC at channel.

Channel	Description
0	-5V monitor
1	Pressure
2	5V Power Supply Voltage
3	5Vanalog Current Monitor
4	3.3V input Current Monitor
5	2.5V input Current Monitor
6	1.8 V input Current Monitor
7	-5V Current Monitor
8	DISC-OneSPE
9	1.8V monitor
10	2.5V monitor
11	3.3V monitor

The ADC channels are described in more detail at
`http://rust.lbl.gov/~gtp/DOM/API/DOM_CPLD_API_v1.0.html`

3.52 rcv

Description

rcv *messageType messageAddress messageLength*

receive a message from the DOR card.

3.53 send

Description

messageType messageAddress messageLength send *send a message to the DOR card.*

3.54 readDAC

Description

channel readDAC *value*

read DAC at channel.

Channel	Description
0	ATWD0 Trigger Bias Current
1	ATWD0 Ramp Top Voltage
2	ATWD0 Ramp Rate Control Current
3	ATWD Analog Reference Voltage
4	ATWD1 Trigger Bias Current
5	ATWD1 Ramp Top Voltage
6	ATWD1 Ramp Rate Control Current
7	PMT Front End Pedestal
8	Multiple SPE Discriminator Threshold
9	Single SPE Discriminator Threshold
10	On-Board LED Brightness Control
11	Fast ADC Reference (Pedestal Shift)
12	Internal Pulser Amplitude
13	Front End Amp Lower Clamp Voltage
14	Spare 10 bit DAD Output 0
15	Spare 10 bit DAD Output 1

The DAC channels are described in more detail at
`http://rust.lbl.gov/~gtp/DOM/API/DOM_CPLD_API_v1.0.html`

3.55 readTemp

Description

readTemp *code*

return coded temperature value

3.56 reboot

Description

reboot

reboot the DOM

3.57 rm

Description

nameAddress nameLength rm

remove the file with name at nameAddress and length of nameLength

Examples

> `s" unused.data" find if rm endif`

Remove a file called unused.data on the flash filesystem

3.58 rshift

Description

value shift rshift *result*

result = value right shifted shift

3.59 s“ words”

Description

s“ words” *wordsAddress wordsLength*

return string at wordsAddress with length wordsLength

3.60 swap

Description

swap

swap the two top words of the stack

3.61 true

Description

true *result*

return a true result (-1)

3.62 type

Description

stringAddress stringLength type

print string at stringAddress with length stringLength

3.63 unlock

Description

nameAddress nameLength unlock

unlock flash data with file name at nameAddress with length nameLength

3.64 usleep

Description

us usleep

sleep for us microseconds

3.65 writeActiveBaseDAC

Description

channel value writeActiveBaseDAC

write value to PMT active base DAC

3.66 writeDAC

Description

channel value writeDAC

write value to DAC channel.

Channel	Description
0	ATWD0 Trigger Bias Current
1	ATWD0 Ramp Top Voltage
2	ATWD0 Ramp Rate Control Current
3	ATWD Analog Reference Voltage
4	ATWD1 Trigger Bias Current
5	ATWD1 Ramp Top Voltage
6	ATWD1 Ramp Rate Control Current
7	PMT Front End Pedestal
8	Multiple SPE Discriminator Threshold
9	Single SPE Discriminator Threshold
10	On-Board LED Brightness Control
11	Fast ADC Reference (Pedestal Shift)
12	Internal Pulser Amplitude
13	Front End Amp Lower Clamp Voltage
14	Spare 10 bit DAD Output 0
15	Spare 10 bit DAD Output 1

The DAC channels are described in more detail at
`http://rust.lbl.gov/~gtp/DOM/API/DOM_CPLD_API_v1.0.html`

Examples

> `1 100 writeDAC`

Set DAC channel 1 to 100 counts

3.67 writePassiveBaseDAC

Description

channel value writePassiveBaseDAC

write value to PMT passive base DAC

3.68 xor

Description

a b xor *result*

result = a xor b

3.69 ymodem1k

Description

ymodem1k *dataAddress dataLength*

Download a file over the terminal return the address and length of the data returned.

4 Coding and Building the software

4.1 Overview

Iceboot is a combination of a few different projects that are checked into the icecube CVS repository. These project are all written in C. The BFD build system (documented elsewhere) is used to build each of these projects. The following sections describe the different projects.

4.2 hal

The hal project defines and implements the hardware access layer. It is used to abstract hardware access to facilitate simulation without real hardware. Hal is documented at `http://deimos.lbl.gov/~arthur/dom-mb`

4.3 dom-loader

The dom-loader project contains the hardware configuration file that is used to configure the hardware on bootup. This allows us to use epxa1 and epxa4 chips and deal with different setting of the dom-mainboard and the epxa eval boards . The config file is parsed to generate the low-level assembler code that configures the hardware.
This project also builds the boot image and contains the low-level code necessary to get read/write access to stdin and stdout over the serial port, or, eventually, the communications channel.

4.4 iceboot

The Iceboot project contains the C source for the forth interpreter.

4.5 stf

Once hardware is working, a pass/fail high-level test is often required. The stf project provides the infrastructure for creating, running, reporting and archiving this test information.

Index